Predictability and Bottlenecks in the Development Flow of Open-Source Projects: an Analysis with Agile Metrics

Maria Eduarda Chrispim Santana
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Brazil
mariae.chrispims@gmail.com

Michelle Hanne Soares de Andrade
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Brazil
michelleandrade@pucminas.br

Cleiton Silva Tavares
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Brazil
cleitontavares@pucminas.br

Abstract

This study investigates predictability and potential bottlenecks in the software development flow of open-source projects. A quantitative analysis was conducted using flow metrics extracted from 3,233 public GitHub repositories. Cycle time, throughput, release interval, and the coefficient of variation were analyzed to assess flow predictability. The results revealed significant differences in metric variability across popularity levels and development stages. In addition, delivery capacity was associated with predictability indicators, particularly cycle time variability.

Keywords

Flow Metrics, Predictability, Bottlenecks, Agile Metrics

1 Introduction

Agile methodologies have become one of the dominant approaches in Software Engineering due to their ability to adapt to dynamic environments and continuous changes [14]. Their adoption has promoted the use of agile metrics, which play a fundamental role in monitoring software processes and evaluating performance [13]. In this context, platforms such as GitHub provide collaborative environments for open-source software development, hosting repositories that store software artifacts such as commits, issues, and pull requests. These artifacts can be explored through software repository mining to extract metrics and analyze software development workflows [19]. From these artifacts, it is possible to derive indicators that characterize the behavior of development activities over time [13]. These indicators constitute flow metrics used to characterize software development processes.

Although agile metrics are widely adopted to monitor software projects and detailed development data are available in repositories such as GitHub, their analysis is often performed in isolation and mainly based on average values, which may limit the understanding of the software development flow [13]. Furthermore, metrics vary across projects due to differences in scope, maturity, and popularity, making it difficult to generalize findings [5]. This scenario becomes even more complex in workflows characterized by high variability and bottlenecks during code review and integration stages [11, 16]. Consequently, it remains insufficiently understood how flow metrics reflect the behavior of software development processes, particularly regarding flow predictability and the identification of potential bottlenecks.

Flow metrics, such as cycle time, throughput, and release interval, are widely used to evaluate performance, delivery capability, and process predictability in software development [18, 20]. However, analyses based solely on average values may conceal important variations across projects and over time. In this context, metric variability becomes a key indicator of process predictability, whereas temporal differences between development stages may indicate potential bottlenecks in the software development flow [11, 12].

Understanding software development flow is essential for supporting decision-making and improving process efficiency, particularly in environments where high variability may compromise delivery predictability and hinder the identification of process inefficiencies. Therefore, investigating the variability of flow metrics and its relationship with predictability and potential bottlenecks may provide deeper insights of software development workflows and support the identification of process inefficiencies.

Based on this motivation, this study aims to identify behavioral patterns, differences across popularity levels, and potential bottlenecks in the software development flow of open-source projects using agile metrics. To achieve this objective, the following specific objectives were defined: (i) analyze the variability of flow metrics across open-source projects belonging to different popularity quartiles; (ii) identify potential bottlenecks by comparing the temporal behavior of development stages; and (iii) investigate the relationship between flow predictability and repository delivery capability.

As its main contribution, this study provides empirical evidence on the relationship between flow metric variability, predictability, and potential bottlenecks through a large-scale empirical analysis of 3,233 public GitHub repositories.

This paper is organized as follows. Section 2 presents the related work. Section 3 describes the materials and methods. Section 4 presents the results on flow metric variability, potential bottlenecks, and the relationship between predictability and delivery capability. Finally, Section 5 concludes the paper.

2 Related Work

Pham and Neumann [13] investigates the use of performance metrics in agile software development through a qualitative case study conducted in an organizational context. The authors analyzed metrics such as Story Points, Velocity, and Lead Time, in addition to software quality and system stability indicators. The results revealed challenges regarding the interpretation, relevance, and consistency of these metrics, mainly due to variability across teams, which hinders comparisons. This study highlights limitations in the use of

software metrics and the influence of variability on development processes, an aspect investigated in the present research through the quantitative analysis of flow metrics in open-source projects.

Flournoy et al. [5] analyzed more than 55,000 observations from 11,398 developers across 216 organizations to investigate factors influencing cycle time in software development. Statistical modeling showed that cycle time exhibits high variability across individuals, teams, and organizations, highlighting the limitations of analyses based solely on average values. These findings reinforce the importance of analyzing flow metric variability, an approach adopted in the present study to investigate behavioral differences across repositories.

Venigalla and Chimalakonda [19] investigated the role of GitHub artifacts in representing software development information. The study combined interviews, a survey, and an exploratory analysis of 950 repositories using topic modeling. The results showed that issues and pull requests contain a substantial amount of process-related information, supporting their use as data sources for extracting flow metrics in the present study.

Tasci and Erdogan [16] analyzed differences in software development workflows between artificial intelligence and traditional software repositories using process mining. The results showed differences in issue closing time and pull request processing time between the analyzed domains, reinforcing the relevance of temporal analyses of software development workflows and the investigation of potential bottlenecks.

Overall, previous studies highlight the relevance of flow metrics for understanding software development processes. This study extends the literature by investigating predictability and potential bottlenecks through a large-scale analysis of 3,233 public GitHub repositories grouped by popularity quartiles.

3 Materials And Methods

This study is characterized as an applied, exploratory, and descriptive empirical study in Software Engineering, adopting a quantitative approach based on metrics extracted from public GitHub repositories [21]. The research investigates the behavior of the software development flow in open-source projects, focusing on flow predictability and the identification of potential bottlenecks. To this end, the method was organized into three main stages: repository collection, data processing, and data analysis.

3.1 Research Questions

To investigate software development flow behavior in open-source projects, focusing on flow predictability and potential bottlenecks, the following research questions (RQs) and hypotheses were defined.

RQ1: Is there a significant difference in the variability (coefficient of variation) of flow metrics across repositories with different popularity levels?

- **H_0:** There is no significant difference in the variability of flow metrics across popularity quartiles.
- **H_1:** There is a significant difference in the variability of flow metrics across popularity quartiles.

This question evaluates whether the variability of cycle time, throughput, and release interval differs across repositories grouped by popularity quartiles based on the number of GitHub stars.

RQ2: Are there significant differences among the temporal stages of the software development flow, indicating potential process bottlenecks?

- **H_0:** There are no significant differences among the temporal stages of the software development flow.
- **H_1:** There are significant differences among the temporal stages of the software development flow.

This question investigates differences among the temporal stages of the software development flow based on the time elapsed between issue creation, pull request integration, and release publication.

RQ3: Is there a significant correlation between flow predictability indicators and repository delivery capability?

- **H_0:** There is no significant correlation between flow predictability indicators and repository delivery capability.
- **H_1:** There is a significant correlation between flow predictability indicators and repository delivery capability.

This question investigates the relationship between flow predictability indicators and repository delivery capability by analyzing correlations between the coefficient of variation of flow metrics and the average repository throughput.

3.2 Experimental Design

The study was conducted using data collected from public GitHub repositories through the GitHub GraphQL API[1] and complementary information obtained from the REST API. GitHub was selected because it is widely adopted by software development teams and provides publicly available artifacts and timestamps required to reconstruct software development workflows. Data collection and processing were implemented in Python using the *requests* and *pandas* libraries[2].

The methodology was structured into three main stages: repository collection, data processing, and data analysis, as illustrated in Figure 1. Each stage comprises specific activities for obtaining and preparing the data required for this study.

Stage 1. Repository Collection. Data were collected through the public GitHub API, using GraphQL for repository search and the REST API for complementary information. An initial set of 10,000 public repositories was collected to reduce sampling bias and increase representativeness, following recommendations from Mining Software Repositories studies [2, 7]. A total of 9,743 valid repositories were obtained.

Due to GitHub search API limitations, repositories were collected in popularity ranges defined by the number of stars, adopted as an indicator of repository popularity [1]. Only repositories with at least 10 stars were considered. Successive queries were executed, with the final range including repositories with more than 50,000 stars. This strategy increased sample coverage and reduced selection bias. Only public, non-archived repositories with issue tracking and

[1] https://docs.github.com/en/graphql
[2] https://pandas.pydata.org/

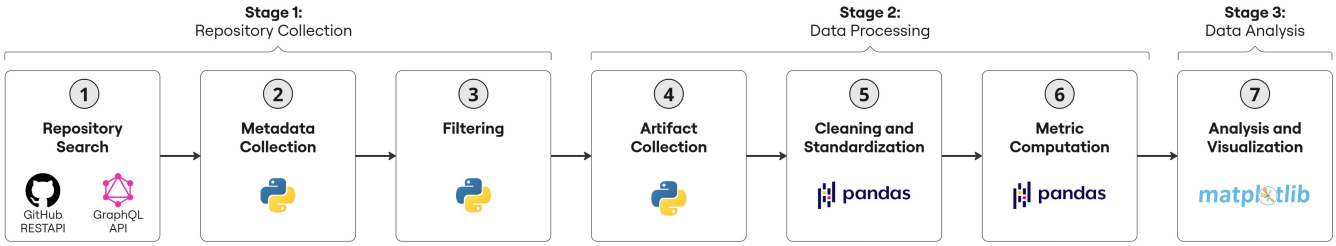

Figure 1: Methodology workflow.

recent activity were considered. Metadata including stars, forks, primary language, issues, pull requests, and releases were collected.

Repositories were filtered using the following criteria: at least two contributors, 15 issues, 10 pull requests, two releases, and activity within the last two years, ensuring active and traceable development workflows. Heuristics based on repository description, topics, and primary programming language were applied to remove non-software repositories, such as educational materials and curated resource lists. The dataset initially comprised 3,249 open-source repositories.

Stage 2. Data Processing. During this stage, detailed software development artifacts were collected, including approximately 4,211,457 issues, 4,699,214 pull requests, and 348,672 releases. For these artifacts, attributes such as creation, closing, merge, and publication timestamps were extracted, enabling the temporal reconstruction of the software development workflow based on observable events, including task creation and completion, code integration, and software release publication.

The collected data were cleaned by removing 16 repositories with inconsistent metadata and collected artifacts, excluding records with missing or invalid timestamps, and standardizing timestamps and measurement units before metric calculation, resulting in a final dataset of 3,233 repositories. Based on the processed data, cycle time, throughput, release interval, and dispersion measures were calculated for each repository.

Stage 3. Data Analysis. During this stage, the calculated metrics were analyzed using descriptive and comparative techniques, including repository segmentation into popularity quartiles based on the number of GitHub stars. Measures of dispersion were used to evaluate flow metric variability and the relationship between predictability and repository delivery capability. Descriptive statistics, including mean and median, were calculated to identify potential bottlenecks across software development stages.

For the statistical analysis, data normality was assessed using the Shapiro–Wilk test. Statistical tests were then applied for group comparisons, correlation analysis, and comparisons among software development stages.

3.3 Flow Metrics

The metrics adopted in this study were defined based on observable events from the software development process, extracted from issues, pull requests, and releases. From these events, process metrics describing the behavior of the software development workflow were calculated.

Development Workflow Decomposition. Due to the absence of an explicit relationship among issues, pull requests, and releases in the collected data, it was not possible to directly calculate the complete lead time. Therefore, the workflow was approximated through observable intermediate stages.

The following intermediate workflow stages were considered:

- time (in days) between issue creation and pull request opening, whenever an explicit relationship between these artifacts existed;
- time (in days) between pull request creation and merge;
- time (in days) between pull request merge and release publication.

The relationship between issues and pull requests was identified using the *closingIssuesReferences* field provided by the GitHub GraphQL API, which records explicit links between these artifacts (e.g., *fixes #123*). The elapsed time between linked artifacts was calculated from their timestamps. Since not all developers use this mechanism, the Issue → Pull Request stage represents only explicitly recorded relationships.

Cycle Time. This metric represents the effective execution time of the development work. It was calculated as the difference between the pull request merge timestamp (*mergedAt*) and its creation timestamp (*prCreatedAt*), as presented in Equation 1. *Cycle time* was measured in days and calculated only for merged pull requests.

$$Cycle\ Time = mergedAt - prCreatedAt \tag{1}$$

Throughput. This metric represents the number of completed work items within a given period. It was calculated as the monthly number of merged pull requests per repository, considering only pull requests with a recorded *mergedAt* timestamp, as defined in Equation 2.

$$Throughput = \frac{Completed\ Items}{Period} \tag{2}$$

***Release* Interval.** This metric measures the elapsed time between consecutive software releases. It was calculated as the difference between the publication date of the current release ($publishedAt_i$) and that of the previous release ($publishedAt_{i-1}$), as defined in Equation 3.

$$Release\ Interval = publishedAt_i - publishedAt_{i-1} \tag{3}$$

To analyze software development flow predictability, the coefficient of variation (CV) was adopted as a measure of dispersion

associated with the flow metrics. It was calculated as the ratio between the standard deviation (σ) and the mean (μ) of the analyzed metric, as defined in Equation 4.

$$CV = \frac{\sigma}{\mu} \tag{4}$$

The interpretation of the coefficient of variation as an indicator of predictability was based on Petersen and Wohlin [12], who argue that higher levels of variability make software development processes more difficult to predict. The CV was adopted because it enables comparisons among repositories with different value scales. Thus, higher CV values indicate lower predictability, whereas lower values suggest greater workflow consistency.

Repositories with a mean value equal to zero produced an undefined coefficient of variation and were therefore assigned a null value. No additional threshold or special treatment was applied to repositories with mean values close to zero.

3.4 Statistical Analysis

Statistical tests were defined to evaluate the hypotheses associated with the research questions. Data normality was initially assessed using the Shapiro–Wilk test [3], allowing the selection of parametric or non-parametric statistical tests. Comparisons among multiple independent groups were performed using ANOVA for normally distributed data [10] or the Kruskal–Wallis test otherwise [15]. Whenever significant differences were identified by the Kruskal–Wallis test, Dunn's post hoc test with Holm correction was applied to perform multiple pairwise comparisons while controlling the Type I error rate [4, 8]. Correlations were evaluated using Pearson's or Spearman's coefficients according to data distribution [17].

4 Results

This section presents the results obtained from the analysis of flow metrics in the analyzed repositories. The first subsection characterizes the analyzed dataset, whereas the remaining subsections present the results associated with each research question.

4.1 Data Characterization

After data cleaning, the final dataset comprised 3,233 open-source repositories, totaling 4,211,457 issues, 4,699,214 pull requests, and 348,672 releases. This dataset enables a large-scale analysis of software development workflows.

Repositories were grouped into four balanced popularity quartiles based on GitHub stars, ranging from 20 to 336,913 stars. The analyzed repositories covered several programming languages, with Python (16.80%), TypeScript (15.09%), Go (10.05%), and JavaScript (9.62%) being the most frequent. Together, the ten most common languages accounted for approximately 81.72% of the repositories.

The average numbers of issues, pull requests, and releases increased substantially from Q1 to Q4, indicating that more popular repositories concentrate higher development activity and software deliveries (Table 1). The median values exhibited the same trend, while the boxplots in Figure 2 revealed considerable dispersion across all quartiles, particularly among the most popular repositories. In the boxplots, the central line represents the median and the marker inside each box represents the mean.

To facilitate visual comparison, outliers were omitted from the boxplots but retained in the statistical analyses. Upper outliers were defined as values greater than $Q3 + 1.5 \times IQR$, and their influence was mitigated through non-parametric statistical tests and relative measures of variability.

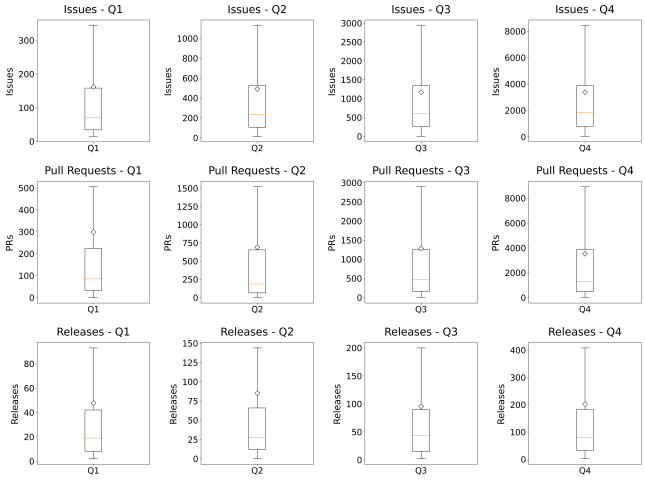

Figure 2: Distribution of issues, pull requests, and releases across popularity quartiles (outliers omitted from plots only).

Flow metrics were calculated for each repository and grouped by popularity quartile. As shown in Table 1, throughput increased from 6.56 to 40.42 pull requests per month from Q1 to Q4, whereas average cycle time slightly decreased from 15.17 to 11.30 days, suggesting higher delivery capability without a proportional increase in integration time.

Table 1: Average values of flow metrics and software artifacts by popularity quartile

Quartile	Issues	PRs	Releases	Cycle Time	Throughput
Q1	163.95	304.47	48.13	15.17	6.56
Q2	477.66	676.56	83.82	17.63	10.47
Q3	1,171.28	1,287.90	95.91	14.40	16.69
Q4	3,393.88	3,547.30	202.71	11.30	40.42

The distribution of flow metrics across popularity quartiles is shown in Figure 3. While cycle time remained relatively stable across quartiles, throughput increased consistently and the release interval decreased in the most popular repositories, indicating more frequent software deliveries. Overall, more popular repositories exhibited greater development activity, delivery capability, and flow metric variability, providing context for the following results.

4.2 RQ1: Flow Metric Variability Across Popularity Quartiles

To answer RQ1, data normality was assessed using the Shapiro–Wilk test applied to the coefficients of variation of throughput, cycle time, and release interval. As shown in Table 2, none of the evaluated metrics followed a normal distribution ($p < 0.05$). High

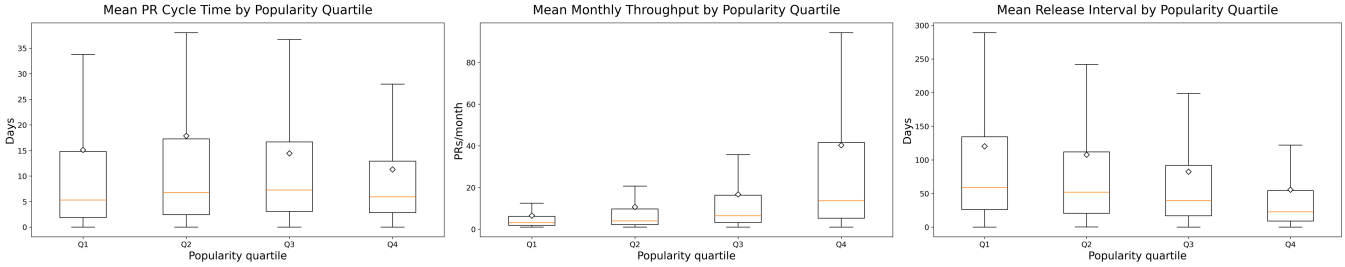

Figure 3: Distribution of flow metrics across popularity quartiles (outliers omitted from plots only).

skewness and substantial differences between mean and median values further supported the adoption of the non-parametric Kruskal–Wallis test to compare the variability of flow metrics grouped by popularity quartiles.

Table 2: Normality test results for RQ1

Metric (CV)	Shapiro–Wilk	p-value
Cycle Time	0.8818	2.77×10^{-44}
Throughput	0.9654	3.55×10^{-27}
Release Interval	0.5925	5.68×10^{-66}

The Kruskal–Wallis test revealed statistically significant differences among popularity quartiles for all evaluated metrics: cycle time ($H = 123.04$, $p < 0.001$), throughput ($H = 36.03$, $p < 0.001$), and release interval ($H = 19.97$, $p < 0.001$). Therefore, the null hypothesis was rejected, indicating that flow metric variability differs across repository popularity levels. Dunn's post hoc test with Holm correction was then applied to identify the groups with significant pairwise differences.

The post hoc analysis revealed statistically significant differences among all quartiles for the coefficient of variation of cycle time. For the coefficient of variation of throughput, significant differences were observed for the pairs Q1–Q3, Q1–Q4, and Q2–Q3. Regarding the coefficient of variation of the release interval, significant differences were identified only for the pairs Q1–Q4 and Q2–Q4. These findings indicate that the variability of flow metrics is not uniformly distributed across repository popularity levels, with the most pronounced differences being observed for cycle time.

4.3 RQ2: Bottleneck Analysis

To answer RQ2, normality was assessed using the Shapiro–Wilk test applied to the temporal stages of the software development workflow. As shown in Table 3, none of the analyzed stages followed a normal distribution ($p < 0.05$). The large differences between mean and median values also indicate skewed distributions affected by extreme values. Therefore, the Kruskal–Wallis test was adopted.

The Kruskal–Wallis test revealed statistically significant differences among the workflow stages ($H = 128878.53$, $p < 0.001$), leading to the rejection of the null hypothesis. The median duration increased from approximately 1 day for PR → Merge to 5 days for Issue → PR and more than 8 days for Merge → Release. Because the data were highly skewed and contained outliers, the interpretation was based on the median. These results suggest a greater temporal concentration after code integration, indicating a potential bottleneck in software delivery.

Table 3: Statistical results for the software development workflow stages

Stage	Shapiro–Wilk	p-value	Mean	Median
Issue → PR	0.3523	5.12×10^{-210}	92.15	4.93
PR → Merge	0.1916	1.85×10^{-217}	10.92	1.09
Merge → Release	**0.2782**	$\mathbf{1.34 \times 10^{-211}}$	**66.14**	**8.31**

These findings suggest a decoupling between code integration and software delivery, indicating that integrated changes may remain staged before being included in a release. According to Humble and Farley [9] and Forsgren et al. [6], the period between integration and release publication may involve automated testing, validation in staging environments, approval processes, and batching changes for software releases. Longer intervals may reflect planned release cycles or delivery strategies rather than process inefficiencies.

4.4 RQ3: Relationship Between Predictability And Delivery Capability

To answer RQ3, data normality was assessed using the Shapiro–Wilk test applied to the predictability indicators and the average monthly repository throughput. The results indicated that none of the analyzed variables followed a normal distribution ($p < 0.05$). Therefore, Spearman's rank correlation coefficient was adopted to evaluate the relationship between flow predictability indicators and repository delivery capability.

The Spearman correlation analysis revealed positive correlations between repository throughput and the evaluated predictability indicators. A moderate positive correlation was observed between the cycle time variability and throughput ($\rho = 0.4435$), suggesting that repositories with greater delivery capability tend to exhibit higher variability in change integration time. In contrast, the correlations between throughput and the coefficients of variation of throughput itself ($\rho = 0.1242$) and the release interval ($\rho = 0.0652$) were classified as weak and very weak, respectively. Overall, the results suggest that the relationship between delivery capability and the evaluated predictability indicators is limited, being more evident for the variability of cycle time. It is important to emphasize that the observed correlations do not imply causal relationships between the analyzed variables, but only indicate statistical associations between repository delivery capability and the considered predictability indicators. Therefore, the null hypothesis was rejected,

indicating evidence of association among the analyzed variables, although the observed effect sizes were predominantly small.

Table 4: Spearman correlation results for RQ3

Variables × Throughput	Spearman ρ	p-value	Interp.
Cycle Time CV	0.4435	1.05×10^{-155}	Moderate (+)
Throughput CV	0.1242	1.41×10^{-12}	Weak (+)
Release Interval CV	0.0652	2.12×10^{-4}	Very weak (+)

4.5 Threats To Validity

The findings should be interpreted considering some limitations. The analyses rely on GitHub API data, and differences in repository practices may affect comparability. The absence of explicit links among issues, pull requests, and releases required the workflow to be decomposed into observable stages. In addition, external factors such as project domain, team size, and development strategies were not considered, and the identified potential bottlenecks may reflect release strategies rather than process inefficiencies. To reduce analytical bias, we used non-parametric tests and relative variability measures like the coefficient of variation. Finally, although non-parametric statistical methods mitigated the effects of non-normality and outliers, the results are limited to the analyzed public GitHub repositories and should be generalized with caution.

5 Conclusion And Future Work

This study investigated the behavior of software development workflows in open-source projects through the analysis of agile metrics extracted from public GitHub repositories. The research was conducted based on three research questions addressing flow metric variability, the identification of potential bottlenecks across workflow stages, and the relationship between predictability and repository delivery capability.

The results revealed significant differences in flow metric variability across repositories with different popularity levels, indicating distinct workflow behaviors. Significant differences were also observed among workflow stages, with the period between pull request merge and release publication presenting the longest median duration, suggesting a potential bottleneck in software delivery. In addition, low-magnitude associations were observed between predictability indicators and delivery capability, with emphasis on the moderate positive correlation between cycle time variability and repository throughput.

This study demonstrates that combining flow metrics with measures of dispersion provides a broader understanding of software development behavior than analyses based solely on average values.

As future work, additional flow metrics, such as Work in Progress (WIP), could be incorporated, as well as factors that may influence process predictability, including programming language, project domain, community size, and collaboration patterns. Future studies may also investigate predictability using probabilistic approaches, such as cycle time percentiles and Monte Carlo simulations, and evaluate different predictability levels across software repositories.

Previous Work

This work was developed as part of an undergraduate thesis in Software Engineering at Pontifical Catholic University of Minas Gerais (PUC Minas).

Artifact Availability

The replication package containing the source code, processed datasets, and scripts used in this study is publicly available at: https://doi.org/10.5281/zenodo.20943924.

References

[1] Hugo Borges, Andre Hora, and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in Open Source Software. *Journal of Systems and Software* 146 (2018), 112–129. doi:10.1016/j.jss.2018.09.016

[2] Hiero Henrique Barcelos Costa, Guilherme Marques de Oliveira, Victor Souza Salles, and Gleiph Ghiotto Lima Menezes. 2017. Tracking the Decisions to Select Repositories for Mining Software Repositories Experiments. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 360–371. doi:10.1109/MSR.2017.42

[3] Wilton de Oliveira Bussab and Pedro Alberto Morettin. 2017. *Estatística Básica* (9 ed.). Saraiva, São Paulo.

[4] Olive Jean Dunn. 1964. Multiple Comparisons Using Rank Sums. *Technometrics* 6, 3 (1964), 241–252.

[5] John C. Flournoy, Carol S. Lee, Maggie Wu, and Catherine M. Hicks. 2025. No Silver Bullets: Why Understanding Software Cycle Time Is Messy, Not Magic. *Empirical Software Engineering* 30 (2025), 174. doi:10.1007/s10664-025-10735-w

[6] Nicole Forsgren, Jez Humble, and Gene Kim. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* IT Revolution Press.

[7] Yiming Gao, Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2014. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 2014 ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 422–431. doi:10.1145/2568225.2568280

[8] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70.

[9] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

[10] Douglas C. Montgomery and George C. Runger. 2010. *Applied Statistics and Probability for Engineers* (5 ed.). Wiley, New York.

[11] Kai Petersen. 2010. An Empirical Study of Lead-Times in Incremental and Agile Software Development. In *New Modeling Concepts for Today's Software Processes*, Jürgen Münch, Ye Yang, and Wilhelm Schäfer (Eds.). Lecture Notes in Computer Science, Vol. 6195. Springer, 345–356. doi:10.1007/978-3-642-14347-2_30

[12] Kai Petersen and Claes Wohlin. 2011. Measuring the Flow in Lean Software Development. *Software: Practice and Experience* 41, 9 (2011), 975–996. doi:10.1002/spe.975

[13] K. P. Pham and M. Neumann. 2024. How to Measure Performance in Agile Software Development? A Mixed-Method Study. In *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Paris, France, 443–450. doi:10.1109/SEAA64295.2024.00074

[14] J. Sadiq and Z. Rana. 2024. The Study of Characteristics of Self Organizing Teams in Agile. In *2024 21st International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. IEEE, 203–208. doi:10.1109/IBCAST61650.2024.10877114

[15] David J. Sheskin. 2003. *Handbook of Parametric and Nonparametric Statistical Procedures* (3 ed.). Chapman and Hall/CRC, Boca Raton.

[16] Oguzhan Tasci and Tugba Gurgen Erdogan. 2025. Comparative Analysis of AI Software and Traditional Software GitHub Repositories Using Process Mining. In *Annals of Computer Science and Information Systems*, Vol. 43. IEEE, 3421–3428. doi:10.15439/2025F3421

[17] Mario F. Triola. 2017. *Introdução à Estatística* (12 ed.). LTC, Rio de Janeiro.

[18] Daniel S. Vacanti. 2016. *Actionable Agile Metrics for Predictability: An Introduction.* ActionableAgile, Seattle, WA.

[19] Akhila Sri Manasa Venigalla and Sridhar Chimalakonda. 2024. An Exploratory Study of Software Artifacts on GitHub from the Lens of Documentation. *Information and Software Technology* 169 (2024), 107425. doi:10.1016/j.infsof.2024.107425

[20] Brennan Wilkes, Alessandra Maciel Paz Milani, and Margaret-Anne Storey. 2023. A Framework for Automating the Measurement of DevOps Research and Assessment (DORA) Metrics. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Bogotá, Colombia, 62–72. doi:10.1109/ICSME58846.2023.00018

[21] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer. doi:10.1007/978-3-642-29044-2